

APPLICATION
FOR
UNITED STATES LETTERS PATENT

TITLE: MULTIPROTOCOL DECAPSULATION/ENCAPSULATION
CONTROL STRUCTURE AND PACKET PROTOCOL
CONVERSION METHOD

APPLICANT: DONALD F. HOOPER AND STEPHANIE L. HIRNAK

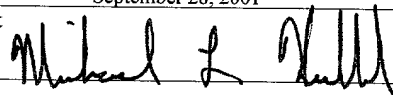
CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL688320799US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

September 28, 2001
Date of Deposit

Signature



Michael L. Hubbard
Typed or Printed Name of Person Signing Certificate

MULTIPROTOCOL DECAPSULATION/ENCAPSULATION CONTROL STRUCTURE
AND PACKET PROTOCOL CONVERSION METHOD

BACKGROUND

This invention relates to forwarding network packets between network domains.

Packets are routed through a series of router devices, each of which stores and forwards packets on its way from a source to a destination. For example, a packet may start out as an Internet packet, be forwarded over an ATM (asynchronous transfer mode path) and then back to Ethernet onto a corporate network to its final intended recipient. As the network passes through these network domains, various header encapsulations may be added to or removed from the packet. Some connections use point-to-point protocol (PPP) whereas others use multiprotocol label switching MPLS, layer to tunneling protocol LTP, ATM and so forth.

DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram of a communication system employing a hardware based multithreaded processor.

FIGS. 2-1 to 2-4 are a detailed block diagram of a hardware based multithreaded processor of FIG. 1.

FIG. 3 is a block diagram depicting a functional arrangement of the multithreaded processor of FIG. 2.

FIG. 4 is a block diagram depicting data structures in memory used for the processor of FIG. 1.

FIG. 5 is a block diagram depicting formats for forwarding tables used in the tables of FIG. 4.

FIG. 6 is a flow chart depicting a generic packet forwarding process.

FIG. 7 is a flow chart depicting an alternative aspect of the packet forwarding process of FIG. 6.

5

DETAILED DESCRIPTION

Referring to FIG. 1, a communication system 10 includes a parallel, hardware-based multithreaded processor 12. The hardware-based multithreaded processor 12 is coupled to a bus such as a PCI bus 14, a memory system 16 and a second bus 18. The system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple microengines 22 each can be simultaneously active and work on multiple hardware controlled threads that independently work on a task.

The hardware-based multithreaded processor 12 also includes a central controller 20 that assists in loading microcode control for other resources of the hardware-based multithreaded processor 12 and performs other general purpose computer type functions such as handling protocols, exceptions, extra support for packet processing where the microengines pass the packets off for more detailed processing such as in boundary conditions. In one embodiment, the processor 20 is a Strong Arm® based architecture. The general purpose microprocessor 20 has an operating system. Through the operating system the processor 20 can call functions to operate on microengines 22a-22f. The

processor 20 can use any supported operating system preferably a real time operating system such as, MicrosoftNT real-time, VXWorks.

The hardware-based multithreaded processor 12 also includes a plurality of function microengines 22a-22f. Functional microengines (microengines) 22a-22f each maintain a plurality of program counters in hardware and states associated with the program counters. Effectively, a corresponding plurality of sets of threads can be simultaneously active on each of the microengines 22a-22f while only one is actually operating at any one time.

In one embodiment, there are six microengines 22a-22f as shown. Each microengines 22a-22f has capabilities for processing four hardware threads. The six microengines 22a-22f operate with shared resources including memory system 16 and bus interfaces 24 and 28. The memory system 16 includes a Synchronous Dynamic Random Access Memory (SDRAM) controller 26a and a Static Random Access Memory (SRAM) controller 26b. SDRAM memory 16a and SDRAM controller 26a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM controller 26b and SRAM memory 16b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and so forth.

The six microengines 22a-22f access either the SDRAM 16a or SRAM 16b based on characteristics of the data. Thus, low latency, low bandwidth data is stored in and fetched from SRAM, whereas higher bandwidth data for which latency is not as

important, is stored in and fetched from SDRAM. The microengines 22a-22f can execute memory reference instructions to either the SDRAM controller 26a or SRAM controller 16b.

Advantages of hardware multithreading can be explained by
5 SRAM or SDRAM memory accesses. As an example, an SRAM access requested by a Thread_0, from a microengine will cause the SRAM controller 26b to initiate an access to the SRAM memory 16b. The SRAM controller controls arbitration for the SRAM bus, accesses the SRAM 16b, fetches the data from the SRAM 16b, and
10 returns data to a requesting microengine 22a-22b. During an SRAM access, if the microengine e.g., 22a had only a single thread that could operate, that microengine would be dormant until data was returned from the SRAM. By employing hardware context swapping within each of the microengines 22a-22f, the
15 hardware context swapping enables other contexts with unique program counters to execute in that same microengine. Thus, another thread e.g., Thread_1 can function while the first thread, e.g., Thread_0, is awaiting the read data to return. During execution, Thread_1 may access the SDRAM memory 16a.
20 While Thread_1 operates on the SDRAM unit, and Thread_0 is operating on the SRAM unit, a new thread, e.g., Thread_2 can now operate in the microengine 22a. Thread_2 can operate for a certain amount of time until it needs to access memory or perform some other long latency operation, such as making an
25 access to a bus interface. Therefore, simultaneously, the processor 12 can have a bus operation, SRAM operation and SDRAM operation all being completed or operated upon by one

microengine 22a and have one more thread available to process more work in the data path.

The hardware context swapping also synchronizes completion of tasks. For example, two threads could hit the same shared resource e.g., SRAM. Each one of these separate functional units, e.g., the FBUS interface 28, the SRAM controller 26a, and the SDRAM controller 26b, when they complete a requested task from one of the microengine thread contexts reports back a flag signaling completion of an operation. When the flag is received by the microengine, the microengine can determine which thread to turn on.

One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a media access controller device e.g., a 10/100BaseT Octal MAC 13a or a Gigabit Ethernet device 13b. In general the network process can interface to any type of communication device or interface that receives/sends large amounts of data. Communication system 10 functioning in a networking application could receive a plurality of network packets from the devices 13a, 13b and process those packets in a parallel manner. With the hardware-based multithreaded processor 12, each network packet can be independently processed. Another example for use of processor 12 is a print engine for a postscript processor or as a processor for a storage subsystem, i.e., RAID disk storage. A further use is as a matching engine. In the securities industry for example, the advent of electronic trading requires the use of electronic

matching engines to match orders between buyers and sellers. These and other parallel types of tasks can be accomplished on the system 10.

5 The processor 12 includes a bus interface 28 that couples the processor to the second bus 18. Bus interface 28 in one embodiment couples the processor 12 to the so-called FBUS 18 (FIFO bus). The FBUS interface 28 is responsible for controlling and interfacing the processor 12 to the FBUS 18. The FBUS 18 is a 64-bit wide FIFO bus, which is currently
10 gaining acceptance as the best bus for Media Access Controller (MAC) devices.

The processor 12 includes a second interface e.g., a PCI bus interface 24 that couples other system components that reside on the PCI 14 bus to the processor 12. The PCI bus
15 interface 24, provides a high speed data path 24a to memory 16 e.g., the SDRAM memory 16a. Through that path data can be moved quickly from the SDRAM 16a through the PCI bus 14, via direct memory access (DMA) transfers. Additionally, the PCI bus interface 24 supports target and master operations. Target
20 operations are operations where slave devices on bus 14 access SDRAMs through reads and writes that are serviced as a slave to target operation. In master operations, the processor core 20 sends data directly to or receives data directly from the PCI interface 24.

25 Each of the functional units are coupled to one or more internal buses. The processor includes an AMBA bus that couples the processor core 20 to the memory controller 26a, 26c and to an AMBA translator 30 described below. The processor also

includes a private bus 34 that couples the microengine units to SRAM controller 26b, AMBA translator 30 and FBUS interface 28. A memory bus 38 couples the memory controller 26a, 26b to the bus interfaces 24 and 28 and memory system 16 including flashrom 16c used for boot operations and so forth.

Referring to FIGS. 2-1 to 2-4, each of the microengines 22a-22f includes an arbiter that examines flags to determine the available threads to be operated upon. Any thread from any of the microengines 22a-22f can access the SDRAM controller 26a, SDRAM controller 26b or FBUS interface 28. The memory controllers 26a and 26b each include a plurality of queues to store outstanding memory reference requests. The queues either maintain order of memory references or arrange memory references to optimize memory bandwidth. For example, if a thread_0 has no dependencies or relationship to a thread_1, there is no reason that thread 1 and 0 cannot complete their memory references to the SRAM unit out of order. The microengines 22a-22f issue memory reference requests to the memory controllers 26a and 26b. The microengines 22a-22f flood the memory subsystems 26a and 26b with enough memory reference operations such that the memory subsystems 26a and 26b become the bottleneck for processor 12 operation.

If the memory subsystem 16 is flooded with memory requests that are independent in nature, the processor 12 can perform memory reference sorting. Memory reference sorting improves achievable memory bandwidth. Memory reference sorting, as described below, reduces dead time or a bubble that occurs with accesses to SRAM. With memory references to SRAM, switching

current direction on signal lines between reads and writes produces a bubble or a dead time waiting for current to settle on conductors coupling the SRAM 16b to the SRAM controller 26b.

That is, the drivers that drive current on the bus need to settle out prior to changing states. Thus, repetitive cycles of a read followed by a write can degrade peak bandwidth. Memory reference sorting allows the processor 12 to organize references to memory such that long strings of reads can be followed by long strings of writes. This can be used to minimize dead time in the pipeline to effectively achieve closer to maximum available bandwidth. Reference sorting helps maintain parallel hardware context threads. On the SDRAM, reference sorting allows hiding of pre-charges from one bank to another bank. Specifically, if the memory system 16b is organized into an odd bank and an even bank, while the processor is operating on the odd bank, the memory controller can start precharging the even bank. Precharging is possible if memory references alternate between odd and even banks. By ordering memory references to alternate accesses to opposite banks, the processor 12 improves SDRAM bandwidth.

The FBUS interface 28 supports Transmit and Receive flags for each port that a MAC device supports, along with an Interrupt flag indicating when service is warranted. The FBUS interface 28 also includes a controller 28a that performs header processing of incoming packets from the FBUS 18. The controller 28a extracts the packet headers and performs a microprogrammable source/destination/protocol hashed lookup (used for address smoothing) in SRAM. If the hash does not successfully resolve,

the packet header is sent to the processor core 20 for additional processing. The FBUS interface 28 supports the following internal data transactions:

FBUS unit	(via AMBA bus)	to/from processor Core.
FBUS unit	(via private bus)	to/from SRAM Unit.
FBUS unit	(via Mbus)	to/from SDRAM.

The FBUS 18 is a standard industry bus and includes a data bus, e.g., 64 bits wide and sideband control for address and read/write control. The FBUS interface 28 provides the ability to input large amounts of data using a series of input and output FIFO's 29a-29b. From the FIFOs 29a-29b, the microengines 22a-22f fetch data from or command the SDRAM controller 26a to move data from a receive FIFO in which data has come from a device on bus 18, into the FBUS interface 28. The data can be sent through memory controller 26a to SDRAM memory 16a, via a direct memory access. Similarly, the microengines can move data from the SDRAM 26a to interface 28, out to FBUS 18, via the FBUS interface 28.

Data functions are distributed amongst the microengines. Connectivity to the SRAM 26a, SDRAM 26b and FBUS 28 is via command requests. A command request can be a memory request or a FBUS request. For example, a command request can move data from a register located in a microengine 22a to a shared resource, e.g., an SDRAM location, SRAM location, flash memory or some MAC address. The commands are sent out to each of the

functional units and the shared resources. However, the shared resources do not need to maintain local buffering of the data. Rather, the shared resources access distributed data located inside of the microengines. This enables microengines 22a-22f, to have local access to data rather than arbitrating for access on a bus and risk contention for the bus. With this feature, there is a 0 cycle stall for waiting for data internal to the microengines 22a-22f.

The data buses, e.g., AMBA bus 30, SRAM bus 34 and SDRAM bus 38 coupling these shared resources, e.g., memory controllers 26a and 26b are of sufficient bandwidth such that there are no internal bottlenecks. Thus, in order to avoid bottlenecks, the processor 12 has an bandwidth requirement where each of the functional units is provided with at least twice the maximum bandwidth of the internal buses. As an example, the SDRAM can run a 64 bit wide bus at 83 MHz. The SRAM data bus could have separate read and write buses, e.g., could be a read bus of 32 bits wide running at 166 MHz and a write bus of 32 bits wide at 166 MHz. That is, in essence, 64 bits running at 166 MHz which is effectively twice the bandwidth of the SDRAM.

The core processor 20 also can access the shared resources. The core processor 20 has a direct communication to the SDRAM controller 26a to the bus interface 24 and to SRAM controller 26b via bus 32. However, to access the microengines 22a-22f and transfer registers located at any of the microengines 22a-22f, the core processor 20 access the microengines 22a-22f via the AMBA Translator 30 over bus 34. The AMBA translator 30 can physically reside in the FBUS interface 28, but logically is

distinct. The AMBA Translator 30 performs an address translation between FBUS microengine transfer register locations and core processor addresses (i.e., AMBA bus) so that the core processor 20 can access registers belonging to the microengines 22a-22c.

The processor core 20 includes a RISC core 50 implemented in a five stage pipeline performing a single cycle shift of one operand or two operands in a single cycle, provides multiplication support and 32 bit barrel shift support. This RISC core 50 is a standard Strong Arm® architecture but it is implemented with a five stage pipeline for performance reasons. The processor core 20 also includes a 16 kilobyte instruction cache 52, an 8 kilobyte data cache 54 and a prefetch stream buffer 56. The core processor 20 performs arithmetic operations in parallel with memory writes and instruction fetches. The core processor 20 interfaces with other functional units via the ARM defined AMBA bus. The AMBA bus is a 32-bit bi-directional bus 32.

Referring to FIG. 3, the multiprocessor 12 is shown performing network routing functions. In one example, an asynchronous transfer mode (ATM), Ethernet and other types of packets enter through the network interface MAC devices and are sent to the network processor 12. These packets are processed in an application on the general purpose microprocessor 20 or on another processor that is coupled through the PCI bus interface (not shown). For reception and transmission of such packets, the application running on that processor 20 or the processor coupled through the PCI bus, makes use of a network stack 72,

which includes network management, control and signaling processes 74 to manage network communications.

The network stack 72 and the application run in the processor 20 that controls the microengines, or another processor coupled to the PCI bus. The paths of receive, transmit and data forwarding represent the transport of the packets through the processor 12. The management control, signaling, and the network stack 72 usually are not involved in data forwarding. Essentially, the processor 20 receives and transmits. The processor 20 generates new packets that are transmitted over the network. The processor 20 can be involved in data forwarding in the exceptional case. This would involve very unusual packets, which may need special handling and complex processing.

For data forwarding processes, the microengines 22a-22f are used. In some instances, data forwarding may occur at the general purpose processor 20 level. The signals Init is programmer's interface for initialization of microengine code. The signal Fini is used for termination (to put control info in a known state). The microengines 22a-22f provide fast, store and forward capabilities. The engines use a multilayer generic look-up process that performs validation, classification, policing and filtering using parallel hardware supported threads of the process. Exceptions and control packets are passed to the processor 20 for processing at the network stack 72. A ternary network stack (not shown) can be located off-chip at a host via the PCI port or device port. This can be used to off-load the processor 20 or centralized management and control for

one place. In some embodiments, the microengine is a compact RISC processor and can have limited instruction space. For this reason and for other reasons, it is desirable to reduce instruction code size when running multiple protocols. The network processor 12 implements a generic forwarding process that can be used to handle various protocol types (both existing and future types) without exceeding instruction storage limits.

Referring now to FIG. 4, an management arrangement 80 for forwarding table structures 90 that are stored in memory is shown. The forwarding table structure management 80 includes a control and management structure 82 including a network stack interface 84 and table managers 86. The table managers 86 manage routing tables 90 that are stored in SRAM and can include a plurality of tables such as shown in FIG. 4 including a layer 4 connection table 92, a layer 3 destination table 94, a layer 2 bridge table 96 and a layer 2 connection table 98.

Additionally, data structures stored in memory can include a packet buffer 100, which is stored in DRAM. The microengines acting as packet data forwarding processors retrieve information from the routing tables 90 in SRAM and store and forward the packet information from the packet buffer in DRAM. The multiple tables 90 are set up by the control management processor 20.

For example, a layer 2 connection table 96 can be used for ATM virtual circuits, frame relay connections MPLS labels or other low level connections. A layer 2 bridge table 96 could be used for Ethernet bridging. A layer 3 destination table 94 could be used for Internet protocol (IP) forwarding based on a destination IP address. The layer 4 connection table 92 could

be used for IP forwarding based on source and destination ports, addresses and protocol. All these tables may require that the packet be decapsulated or encapsulated.

Once the tables 90 are populated with forwarding information in a generally conventional manner, packet data forwarding processors can receive packets, perform table look-ups to obtain information and convert packets as required by the table entry. The control management process sets up the tables 90 with a common format for the purpose of decapsulation and encapsulation.

Referring now to FIG. 5, exemplary table entries, a subset of which are included in each of the tables 90, is shown. The table entries include the following fields:

Forwarding Table Format

Decap Flag. Indicates whether the bytes should be stripped from the packet. If this flag is asserted, then the number of bytes to strip is in Decap Byte Count field.

Decap To Layer. This field specifies decapsulation of header layers up to the specified layer. The length of the layer and hence the decapsulation is determined by parsing the packet header.

Decap Byte Count. This field specifies the number of bytes to remove from the front of the packet. Decap is performed by adjusting the packet start offset in the packet buffer.

Current Encap. This field specifies an identifier of the current packet encapsulation type.

Encap Flag. Indicates whether bytes should be prepended to the packet. If this flag is asserted, then the number of bytes is in Encap Byte Count field, and the bytes to be encapsulated is in the Encap Header field.

Encap Byte Count. Number of bytes to be prepended to the packet.

Encap Header. The actual bytes to be prepended.

Next Table Type. If non-zero, this indicated that a further lookup is required. This gives the table type. For example, layer 3 routing or layer 4 connection table type. A layer 3 routing lookup would use a longest prefix match lookup algorithm using the destination IP address. A layer 4 connection lookup would use a 104 bit hash algorithm using source and destination addresses, source and destination ports, and protocol.

Next Table Addr. There can be multiple next tables, and multiple next tables of the same type. This field specifies the base address of the table.

The flags get set or cleared by the management process. Signaling and setting up connections are part of the network system that will determine that a certain path through the network requires a change of the header. There can be many reasons why a header can change. Usually a header change is

used when the protocol changes from one network domain to another.

Referring now to FIG. 6, a process 110 for encapsulating/decapsulating generic protocols is shown.

5 Initially one of the microengines 22a-22f receives 112 a packet from the network interface. The packet is comprised of one or more headers followed by a payload. The microengine, e.g., 22a copies the payload portion of the packet to a packet buffer in DRAM and it may place the packet at an offset in the buffer to
10 make room for any new header that could be prepended to the packet for packet forwarding. The packet offset parameter for that packet is set to a default value determined at the offset into the buffer. The microengine reads 114 in the first header of the packet and performs a layer 2 look-up. The layer 2 look-up will read the table layer 2 bridge table and/or layer 2
15 connection table. The tables will return various parameters such as decap or encap flags. The process 110 will determine 116 if the decap or encap flags are set. If the decap and encap flags are set, the process will add 118 the decap byte count to
20 the packet start offset and will subtract 120 the encap byte count from the packet start offset and prepend the encap bytes to the packet. The process 110 tests 122 if there is a next table to examine by looking at the blank field in the currently read table. If there is a next table, the process 110 will
25 parse the next header 124, fetch and read the next table. The process 110 continues looking to test the decap or encap flags being set.

If, however, the process did not determine that the decap and encap flags were set (116, above), it would determine 130 if the encap flag or the decap flag were set 132. If the encap flag was set, it will subtract 120 the encap flag byte count from the start offset and prepend the encap bytes to the packet. On the other hand, if the decap flag was only set 132, the process will add 134 a decap byte count to the buffer offset and, in any event, will check the next table 112. When the process determines that it is at the end of checking the tables, it will then classify and forward 136 the packet in a conventional manner. That is, the "no" condition indicates that the process can classify and forward. Forwarding the header can have the microengine take the header and send it to the processor 20 or elsewhere, so that it can get reassembled with the payload. Forwarding the header could also involve forwarding the packet, etc.

Referring now to FIG. 7, in addition to specifying byte dissemination counts obtained from the look-up table. The look-up table may have the decap to layer field set in the table.

This field specifies that the front portion of a packet should be decapsulated up to a certain layer. As known, packets are defined in protocol layers used in the OSI (Open Systems Interconnect) seven layer network protocol. After passing through physical layer 1, the first software layer seen by the network processor layer is layer 2, also referred to as the link layer. The length of the bytes to be decapsulated is determined by parsing the packet layers prior to the layer that is to be

the new start of the packet. The length can be added to the packet start offset.

FIG. 7 shows a variation where the decapsulation length is not specified in the table, but is determined by reading the packet itself. In other words, this would be a set of routines that would be inserted into the processing of FIG. 6 substitute encapsulated byte count from the packet into the offset.

A process 140 to determine this offset is shown in FIG. 7. The process 140 includes reading the table 142, determining that the decap to packet layer bit 144 has been set, and if set, retrieve the length of the layer to be removed by parsing the header 146 and adding the length to the packet start offset 148. If the decap layer has not been set then the process simply skips. In any event, this process can be prepended to the process described in conjunction with FIG. 6.

A typical use of a decap to layer bit is to specify a decapsulation up to the layer 3 IP header. If the packet encapsulation is a multiprotocol over an ATM network such as the RFC 1483 standard, the layer 2 header length is determined by parsing the layer to header itself using the RFC 1483 length rules. However, if the packet encapsulation is classical IP the layer 2 length is determined by following the classical IP layer length rules. The packet encapsulation may be known by the port type it came in on from the prepended custom header from that port or may be obtained from the first look-up table in the current encap field.

Rather than each network protocol defining a separate protocol conversion this technique provides a generic approach.

The approach saves code space and software development time-to-market. In an alternative embodiment, this technique can be implemented as a software library routine, e.g., a generic software building block for decapsulation/encapsulation, where
5 customers can insert their proprietary header encapsulation and a customer's vendor need not get involved with customer's proprietary protocol designs.

A number of embodiments of the invention have been described. Nevertheless, it will be understood that various
10 modifications may be made without departing from the spirit and scope of the invention. Accordingly, other embodiments are within the scope of the following claims.